Optimization and Equivalence Checking

Christian Krieg

November 5, 2024

Welcome to this year's *Digital Integrated Circuits* course! Throughout this course, we will have to solve some assignments that will teach us fundamental concepts of integrated circuit design.

We will demonstrate digital integrated circuit design on a step-by-step simple example of a digital counter. As the name of this course suggests, our ultimate goal is to implement this counter as a digital integrated circuit. In the lab, we map this counter to a field-programmable gate array (FPGA) architecture, and run it on the board shown in Figure 1. The counter's functionality is executed on the FPGA. We will be able to control execution by pushing the buttons, and to visualize the counter value using the light-emitting diodes (LEDs). Because operating a hardware counter with buttons and LEDs may be inconvenient, we also will interface the hardware counter over a universal asynchronous receiver/transmitter (UART), so that we may be able to further process the counter value on a host computer. But this will all happen during the lab week at the end of the exercises.

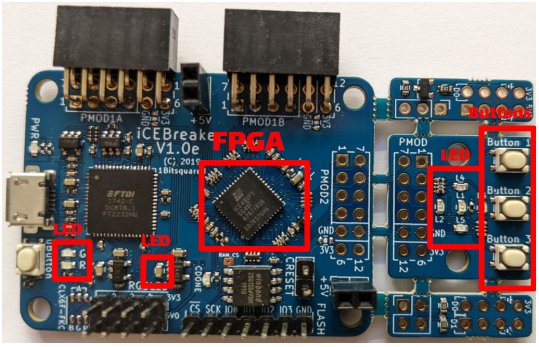

Figure 1: The *icebreaker* board, which will run our counter

Before we can bring our counter on hardware, it is important that we learn how to design digital circuits using state-of-the-art (and beyond) methods and tools, and to get a feeling for the results of our design decisions. We will draw attention on the results of the synthesis steps, which we will verify with several tools along the digital design flow.

Generally speaking, we perform the following steps:

1. Specify the counter's behavior
2. Model (implement) the counter in a hardware description language (HDL)
3. Verify the counter's implementation
4. **Synthesize and optimize the counter's HDL model + verify**
5. Map the counter to the target (FPGA) technology + verify
6. Generate an FPGA bitstream + verify
7. Configure the FPGA with the bitstream + verify
8. Run the counter on hardware

Our overall task is to build a synchronous counter that runs in two modes, based on a *mode* switch. The two modes are called *up* and *down*. In *up* mode, the counter increments the counter value by **5** at the rising edge of the clock signal, and in *down* mode, it decrements the counter value by **9** at the rising edge of the clock signal. The counter should implement synchronous reset behavior, such that it is set to an initial value of **-50** upon reset. The counter should never exceed a value of **235**, and should never go below a value of **-230** (the counter value sticks near the minimum/maximum value, until the counting direction changes). The counter value should never be **-11** (the counter should jump over this value by one increment/decrement).

Listing 1: CLA implementation (*map.v*)

```
//------------------------------------------------------------------------------
//
// Maps $add cell to a carry-lookahead adder implementation
//
module \$add (A, B, Y);

   parameter A_WIDTH = 32;
   parameter B_WIDTH = 32;
   parameter Y_WIDTH = 32;
   parameter A_SIGNED = 0;
   parameter B_SIGNED = 0;

   input [A_WIDTH-1:0] A;
   input [B_WIDTH-1:0] B;
   output [Y_WIDTH:0] Y;

   wire [Y_WIDTH:0] G;
   wire [Y_WIDTH:0] P;
   wire [Y_WIDTH:0] C;

   // Generate CARRY logic
   // i: Output bit index
   // j: Term index
   // k: Carry input signal index
   genvar i, j, k;
   generate
      // Calculate carry bit for each output bit
      for (i = 0; i <= Y_WIDTH; i++) begin
         wire [i-1:0] carry_terms;
         assign C[i] = |carry_terms;
         // Calculate terms for each carry bit
         for (j = 0; j < i; j++) begin
            wire [i-1:j] carry_in;
            assign carry_terms[j] = &carry_in;
            // Connect input signals for each term of the carry bit
            assign carry_in[j] = G[j];
            for (k = j+1; k < i; k++)
               assign carry_in[k] = P[k];
         end
      end
   endgenerate

   assign G = A & B;
   assign P = A ^ B;

   assign Y[0] = P[0];
   assign Y[Y_WIDTH] = C[Y_WIDTH];
   assign Y[Y_WIDTH-1:1] = P[Y_WIDTH-1:1] ^ C[Y_WIDTH-1:1];

endmodule
//
//------------------------------------------------------------------------------
```

Wrapping up: carry-lookahead adder implementation

In the last task, we learned how to define a low-level implementation for the high-level (arithmetic) addition operation, and to use the *techmap* command to automatically replace this operation by the desired low-level implementation. We implemented a carry-lookahead adder (CLA), and mapped it to the addition operation of our behavioral counter from Task 4.

Now, the cool part: We used the assertions specified in Tasks 1 and 2 to prove the functional correctness of our counter. While we significantly modified the implementation of a major part of our counter (the addition), we were not required to change our verification methodology. When passing the formal verification, we know that our counter still behaves as specified.

In case you did not yield a working solution for Task 4, Listing 1 provides a sample CLA implementation. Figure 2 shows an implementation of a 4-bit CLA, with the corresponding Boolean equations, with P_i being the *propagate* term, G_i the *generate* term, and C_i the carry for bit i:

$$C_1 = G_0 + P_0 \cdot C_0 \tag{1}$$
$$C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \tag{2}$$
$$C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \tag{3}$$
$$C_4 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3 \tag{4}$$

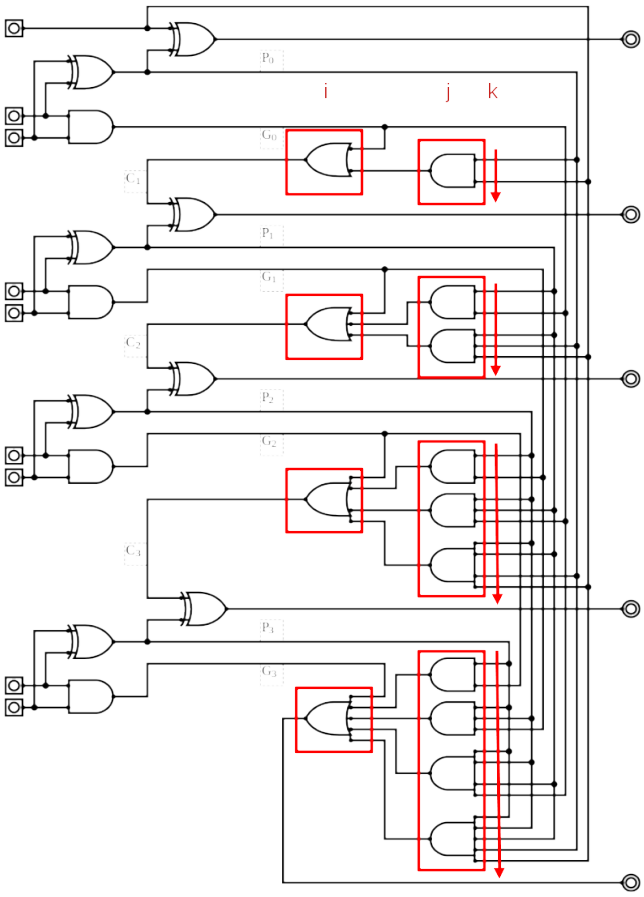

Figure 2: 4-bit CLA implementation (adapted from https://upload.wikimedia.org/wikipedia/commons/1/16/Four_bit_-adder_with_carry_lookahead.svg)

In Listing 1, we define *generate variables* i, j, k (Line 25) in order to generate the carry logic in the *generate* block given in Lines 26 to 41. Figure 2 visualizes the scopes of i, j, k, where i iterates over the adder's carry bits (Lines 28 to 40), j iterates over the (conjunction) terms that contribute to each carry bit (Lines 32 to 39), and j iterates over the input signals that contribute to these conjunctions (Lines 37 to 38), as given in Equations (1) to (4). Because C_0 is not defined for our adder, we ignore it in Listing 1. The conjunctions are modeld in Line 34, using the *reduction and* operator. The carry bits are modeled in Line 30, using the *reduction or* operator. In Lines 43 and 44, the *generate* and *propagate* terms are defined. The adder's output is defined in Lines 46 to 48.

Optimizing our counter

In this task, we will learn how to optimize our register transfer level (RTL) counter using *Yosys*, and how to prove that the optimized version of our counter is functionally equivalent to our un-optimized counter.

Optimization

We already know how to read a design, and also how to synthesize processes and map structural implementations. Now we will learn how to optimize our design using the *opt* command. Have a look at the *opt* command's help text (by calling `help opt` from within *Yosys*) in order to find out what we can do with Yosys's built-in optimization. Unless we do not want to do something highly-special, for the most cases a simple `opt -full` will optimize our design. Write the optimized version of our counter to a Verilog file named *counter_opt.v* (Listing 2).

Again, use *show* and *stat* after each step to get a feeling about the impact of *opt -full*. What do we notice?

Equivalence checking

Having applied optimization techniques to our counter design, the question is now: How can we be sure that the counter still behaves as originally specified? Of course, we still have our properties at hand that specify our counter. And we

Listing 2: Optimizing our counter

```
# Read the Verilog model of the counter
read_verilog counter.v

# Prepare the design (elaborate hierarchy, synthesize processes, etc)
prep -auto-top
clean

# Map the CLA implementation
techmap -map map.v

# Optimize the design
opt -full

# Rename the module and write it to a Verilog file
rename counter counter_opt
write_verilog counter_opt.v
```

Listing 3: Creating a miter circuit for equivalence checking

```
# Read the 'gate' model
read_verilog counter_opt.v

# Read the 'golden' model
read_verilog counter.v

# Synthesize processes and clean the design
proc
clean

# Create the miter
miter -equiv -make_assert counter counter_opt equal
```

can still use them to functionally verify the optimized implementation of the counter against its specification.

However, there is another verification technique that allows us to compare two designs, and make a decision whether these two designs are functionally equivalent: This verification technique is called *equivalence checking*. Yosys offers equivalence checking capabilities, implemented in two commands: *miter* for generating a circuit to check two designs for equivalence, and the *sat* command to encode and solve the problem of finding out functional equivalence between the two designs.

Your task

Listing 3 shows the flow to create a miter circuit for equivalence checking. The terminology in equivalence checking says that we check a *gate* model against a *golden* model. To understand how a miter circuit works, we perform the *Yosys* commands given in Listing 3. Next, we show the miter circuit using *show equal*. We will see two cells, *gold: counter* and *gate: counter_opt*. They are provided the same inputs, and they feed their outputs into a comparator which indicates whether the gates' outputs are equal. The comparator's output, which is called the *trigger* output, is connected to an *assert* cell's input. The *assert* cell is created because we issue *-make_assert* to the *miter* command, and it checks that the trigger output is always low.

Next, it is time to do the actual checking. Listing 4 shows how we can prove the equivalence between our optimized and un-optimized counter. However, we need to flatten the miter circuit before checking it (We should be able to do this on our own ;). The *sat* command given in Listing 4 checks all assertions (*-prove-asserts*) in the design (remember the *assert* cell in the miter circuit). It checks that the assertions hold forever by performing a *temporal induction* proof (*-tempinduct*). The *-verify* option tells Yosys to exit once an assertion is violated. This is important, because if we automatically check a design using a script, we want to know when an assertion is triggered. The *-show-regs* option tells the *sat* command to show values for all registers in the design when creating a counterexample for any violated assertion. In our case, we have only one register, the *cnt* register that holds the counter value. The *-dump_vcd* option tells the *sat* command to create a signal trace to reproduce the error that lets an assertion fail. Finally, we specify the module to check. In our case, we want to check the miter module (*equal*).

Listing 4: Command to prove equivalence using the miter circuit specified in Listing 3

```
sat -prove-asserts -tempinduct -verify -show-regs -show-inputs -show-outputs -dump_vcd trace.vcd equal
```

Listing 5: Commands to inject a fault into our optimized counter

```
read_verilog counter_opt
proc
clean
rename counter_opt counter_buggy
cd counter_buggy
ls
delete $xor$counter_opt.v:<here comes what is specific to your design>
clean
write_verilog -selected counter_buggy.v
```

The *opt* command is a well-tested command and it is very unlikely that the equivalence check returns something different than *QED*, which means that the two versions of our counter behave functionally equivalent.

However, we want to get a feeling about the power of the *sat* command. We therefore inject a fault into the optimized version of our counter. Let us simply delete a cell from it. This will introduce different behavior which will be detected by the *sat* command. Listing 5 shows the commands necessary to inject such a fault. We change into the optimized counter's module (*cd counter_opt*), and list all cells and wires. Let's delete the last cell displayed by *ls*, which will be something like `$xor$counter_opt.v:306$97`. Delete that cell using the *delete* command. Yosys supports *tab completion*, so instead of typing every single character that specifies the name of our last cell, we simply type `$xor` and then hit the tabulator key on our keyboard. Next, clean the design from disconnected wires that results from deleting the cell. Finally, use *write_verilog* again to write the buggy version of our counter into a Verilog file called *counter_buggy.v*.

Repeat the steps given in Listing 3 and Listing 4 in order to see what happens if the two designs being checked are not functionally eqiuvalent. Replace *counter_opt* by *counter_buggy*, and add the *-flatten* option to the *miter* command, such that we do not explicitly have to flatten the miter.

Have a look at the trace file! What we will see is quite amazing: The counter behaves correctly for a number of clock cycles, counting up and down while producing correct results. But after some clock cycles the buggy counter's behavior is different from the expected behavior.

We can see here that test results strongly depend on the selection of test vectors. A bug is only captured if the correct sequence of input patterns is provided to a design in order to exhibit buggy behavior. This is why verification by simulation is very error-prone.

As we can see, formal verification (what we do here) is very powerful, and can capture such bugs easily!

Submission details

Attach your *counter.v*, *counter_opt.v* and *counter_buggy.v* to your submission e-mail.